Department I - C Plus Plus

Modern and Lucid C++
for Professional Programmers

Week 2 – Values and Streams

Thomas Corbat / Felix Morgner
Rapperswil, 25/28.09.2023
HS2023

Goals

- You know how and where to define variables
- You can identify the type of a literal
- You know the most important operators
- You can use the basic sequence container `std::string`
- You can read from and write to streams
- You know about the possible states of an `std::istream`

Topics

- **Variable Definitions**
- **Values and Expressions**
- **Strings and Sequences**
- **Input and Output Streams**

Recap Week 1

Ostschweizer Fachhochschule

- **What is the output?**

Point.hpp
```
struct Point {
  int x;
  int y;
};
auto modify(Point in) -> Point {
  in.x = 3;
  in.y = 4;
  return in;
}
```

main.cpp
```
#include "Point.hpp"
#include <iostream>
auto main() -> int {
  Point point{1, 2};
  Point other = modify(point);
  std::cout << '{' << point.x << '/' << point.y << "}\n";
  std::cout << '{' << other.x << '/' << other.y << "}\n";
}
```

- **How do you prevent the violation of the One Definition Rule?**

```cpp
struct Point {
  int x;
  int y;
};
```

Point.hpp

```cpp
#include "Point.hpp"
struct Square {
  Point p1;
  Point p2;
};
```

Square.hpp

```cpp
#include "Square.hpp"
#include "Point.hpp"
auto main() -> int {
  Square s{Point{1, 2}, Point{9, 9}};
  //...
}
```

main.cpp

Variable Definitions

Goals:

- You know how and where to define variables.
- You want to make all your variables const

Defining Variables

```
<type> <variable-name>{<initial-value>};
```

Examples

```
int anAnswer{42};
int zero{};
```

- **Defining a variable consists of specifying its `<type>`, its `<variable-name>` and its `<initial value>`**

- **Using = or {} for initialization with a value supplied we can have the compiler determine its type**

```
auto const i = 5;
```

- **Initialization might be omitted but that is bad practice and potentially dangerous**

```
double x;
```

```
int const theAnswer{42};
```

- **Adding the `const` keyword in front of the name makes the variable a single-assignment variable, aka a constant**
  - A `const` variable must be initialized

```
int const theAnswer;
```

  - A `const` variable is immutable

```
theAnswer = 15;
```

```
theAnswer++;
```

- **Some constants are required to be fixed at compile time (Topic in C++ Advanced)**
  - To enforce that use the keyword `constexpr`

```
double constexpr pi{3.14159};
```

- **The keyword `const` also appears in other contexts**
  - It always denotes something immutable (there is also a `mutable` keyword).

You should use `const` whenever possible for non-member variables!

- **Why should I use const?**
  - A lot of code needs names for values, but often does not intend to change it
  - It helps to avoid reusing the same variable for different purposes (code smell)
  - It creates safer code, because a const variable cannot be inadvertently changed
  - It makes reasoning about code easier
  - Constness is checked by the compiler
  - It improves optimization and parallelization (shared mutable state is dangerous)
- **Computing values and functions only without side-effects is possible and a specific style supported by C++**

- **As close to its use/as late as possible**
  - Do not practice to define all (potentially) needed variables up front (that style is long obsolete!)
  - More chances for "`const`"-ness: single assignment and `auto` declaration

```
auto readAverage() -> int {

  int sum = 0;
  int count = 0;

  while (std::cin) {
    auto const next = readInt();
    sum += next;
    count++;
  }
  return sum / count;
}
```

```
auto readAverage() -> int {
  int next;
  int sum = 0;
  int count = 0;

  while (std::cin) {
    next = readInt();
    sum += next;
    count++;
  }
  return sum / count;
}
```

- **Every mutable global variable you define is a design error!**
  - Code using (non-const) globals is almost untestable
  - Concurrent code with globals requires careful synchronization!

```
int sum = 0;
int count = 0;

auto readAverage() -> int {
  while (std::cin) {
    sum += readInt();
    count++;
  }
  return sum / count;
}
```

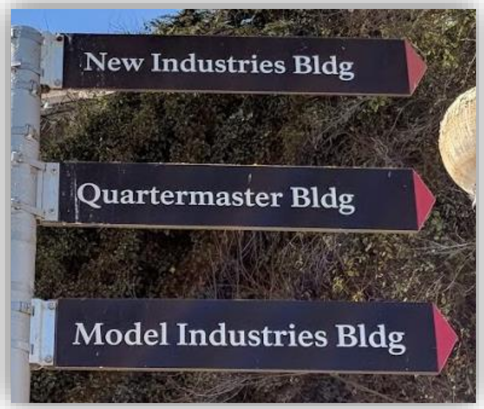
- **Scoping rules are similar to Java's:**
  - A variable defined within a block is invisible after the block ends
  - Difference: Avoid name clashes, i.e., redefining an existing variable inside a block is not an error in C++

```
auto process(int value) -> void { //#1
  //…
  {
    int value; //#2
    //…
    {
      int value; //#3
      //…
    } //lifetime of #3 ends here

  } //lifetime of #2 ends here

} //lifetime of #1 ends here
```

A Note on Naming



- **The C++ convention is to begin variable names with a lower-case letter**
- **Spell out what the variable is for**
- **Do not abbreviate `uncsrly`**

```
int const mpm = 1609;
```

```
int const metersPerMile = 1609;
```

- **Very short (one letter) names can be used in tightly bound context, e.g., for iteration indices or very short scopes**

```
for (auto i = 0; i < size; i++) {
  //...
}
```

- **C++ has a whole bunch of built-in types, mostly for numbers**
  - They are part of the language and don't need an `#include`
  - **`short, int, long, long long –`** each also available as **`unsigned`** version
  - **`bool, char, unsigned char, signed char`**
    - They are treated as integral numbers as well
  - **`float, double, long double`**
  - **`void`** is special, it is the type with no values
  - Plus some more, not relevant now
- **The standard library provides a multitude of types for different purposes (defined in classes)**
  - Important: **`std::string`** and **`std::vector`**
  - Their use requires **`#include`** of the type definition

Values and Expressions

Ostschweizer Fachhochschule

Goals

- You can identify the type of a literal
- You know the most important operators

Literal Value Examples

Literal Example	Type	Value

`'a'`
`'\n'`
`'\x0a'`		

`1`
`42L`
`5LL`
`int{} (not really a literal)`		

`1u`
`42ul`
`5ull`		

`020`
`0x1f`
`0XFULL`		

`0.f`
`.33`
`1e9`
`42.E-12L`
`.3l`		

`"hello"` `"\012\n\\"`		

Literal Value Examples

Literal Example	**Type**	**Value**

`'a'`
`'\n'`
`'\x0a'`
	char
	char
	char
		Letter a, value: 97
		<NL> character, value: 10
		<NL> character, value: 10

`1`
`42L`
`5LL`
`int{}` (not really a literal)
	int
	long
	long long
	int
		1
		42
		5
		0 (default value)

`1u`
`42ul`
`5ull`
	unsigned int
	unsigned long
	unsigned long long
		1
		42
		5

`020`
`0x1f`
`0XFULL`
	int
	int
	unsigned long long
		16 (octal 20)
		31 (hex 1F)
		15 (hex F)

`0.f`
`.33`
`1e9`
`42.E-12L`
`.3l`
	float
	double
	double
	long double
	long double
		0
		0.33
		1000000000 (10^{9})
		0.000000000042 ($42*10^{-12}$)
		0.3

`"hello"` `"\012\n\\"`	char const [6] char const [4]	Array of 6 chars: h e l l o <NUL> Array of 4 chars: <NL> <NL> \ <NUL>

- **Arithmetic**
  - binary: `+ - * / %`(modulo)

```
a + b
```

  - unary: `+ - ++ --`

```
++e
```

- **Logic**
  - ternary/conditional: `?:`
  - binary: `&& and || or`
  - unary: `! not`

```
(!a || b) ? c : d
```

- **Bit-operators**
  - binary: `& | ^ << >> bitand bitor xor`
  - unary: `~ compl`
  - Use unsigned types of bit operators

```
flags & mask
```

- `(5 + 10 * 3 - 6 / 2)`

- **Precedence as in normal mathematics**
  - `5 + 30 - 3 => 32`

Complete list for operator precedence on CppReference:
https://en.cppreference.com/w/cpp/language/operator_precedence

- `auto x = 3 / 2;`

- **Fraction results of integer operations are always rounded down (towards zero)**
  - `3 / 2 => 1`

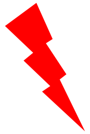
- `auto y = x % 2 ? 1 : 0;`

- **Integer to boolean conversion**
  `0 -> false` / every other value `-> true`
  - `true ? 1 : 0 => 1`

Caution: Type Conversion

- **C++ provides automatic type conversion if values of different types are combined in an expression**
  - Unless in braced initialization

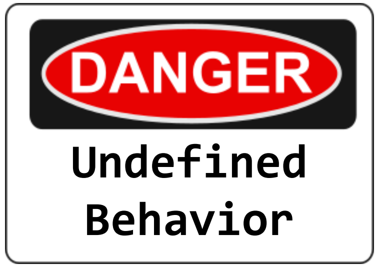
```
int i{1.0};
```



- **Division of integers does not round**
  - **`double x = 45 / 8;`**

Value of `x: 5`

- **Dividing integers by zero is undefined behavior**
  - that is also true, when using modulo `(5%0)`



left: lvalue

```
x = 6 * 7;
```

right: rvalue

- **Assignment requires a variable on the left side: an lvalue (x)**
  - Elements in a container can also act as an lvalue

```
greeting.at(1) = 'H'
```

- **The value on the right side is an rvalue**
  - `6 * 7`
- **Most binary operators can be combined with assignment to shorten the code**
  - `a += b;  //a = a + b`
  - `c /= d;  //c = c / d`
  - `x >>= 2; //x = x >> 2`

- **Relational Operators compare values**
  - `< > <= >= == !=`
  - results in `true` or `false`
- **Logical operators and conditional statements are generous to accept numeric values as statement of truth**

```
if (5);
while (1);
std::cout << (!x % 2 ? "even" : "odd ");
if (a < b < c);
```

Compiles, but what does it mean?

- **C++20 features three-way-comparison: <=> (more on that later)**

- **Use `double` - usually most efficient on current hardware and default for floating point literals**
  - Use `float` only, if memory consumption is utmost priority (very very large data sets) and precision and range can be traded (on 64bit often not beneficial)
- **Remember there are legal double values that are not numbers:**
  `NaN`, `+Inf`, `-Inf` (not-a-number, plus/minus infinity)
- **Comparing floating points for equality (==) is usually wrong**
  - CUTE's `ASSERT_EQUAL(expected, actual)` automatically provides a “delta" value as a margin to consider almost equal values equal
  - Or use `ASSERT_EQUAL_DELTA(expected, actual, delta)`

Strings

Ostschweizer Fachhochschule

Goals:

- You know the basic sequence container `std::string`

- **`std::string` is C++'s type for representing sequences of char (which is often only 8 bit)**
  - Unicode support is different from Java (Advanced C++)

```
std::string name{"Carl"};
```

String Literals

- **Literals like "ab" are not of type `std::string`**
  - Array of const characters
  - Null terminated

'a'	'b'	\0

  - Type: `char const[3]`
- **But "ab"s is an `std::string`**
  - Requires `using namespace std::literals;`
  - Important when types are deduced (`auto` variables, templates)

```
auto printName(std::string name) -> void {
  using namespace std::literals;
  std::cout << "my name is: "s << name;
}
```

- **`std::string` objects are mutable in C++**
  - In Java String objects cannot be modified
- **You can iterate over the contents of a string**
  - With iterators (or loops if necessary)

```
auto toUpper(std::string & value) -> void  {
  for (int i = 0; i < value.size(); i++) {
    value[i] = toupper(value[i]);
  }
}
```

```
auto toUpper(std::string & value) -> void {
  for (char & c : value) {
    c = toupper(c);
  }
}
```

```
auto toUpper(std::string & value) -> void {
  transform(cbegin(value), cend(value), begin(value), ::toupper);
}
```

For a complete list of capabilities see: https://en.cppreference.com/w/cpp/string/basic_string
`<cctype>` include for `toupper()`

```cpp
#include <iostream>
#include <string>

auto askForName(std::ostream & out) -> void {
  out << "What is your name? ";
}

auto inputName(std::istream & in) -> std::string {
  std::string name{};
  in >> name;
  return name;
}

auto sayGreeting(std::ostream & out, std::string name) -> void {
  out << "Hello " << name << ", how are you?\n";
}

auto main() -> int {
  askForName(std::cout);
  sayGreeting(std::cout, inputName(std::cin));
}
```

Input and Output Streams

Goals:

- You know how to read and write from and to streams
- You know about the possible states of an `std::istream`
- You can read input from an `std::istream` safely

```
auto askForName(std::ostream & out) -> void
```

```
auto readName(std::istream & in) -> std::string
```

- **`std::string` and built-in types represent values**
  - Can be copied and passed-by-value
  - No need to allocate memory explicitly for storing the chars
- **Some objects aren't values, because they can not be copied:**
  - Streams representing the program's I/O
- **Functions taking a stream object must take it as a reference, because they provide a side-effect to the stream (i.e., output characters)**

- **Stream objects provide C++'s I/O mechanism**
  - Pre-defined globals: `std::cin` `std::cout` 😟
- **Use them ONLY in the `main()` function!**
- **"shift" operators read into variables or write values**
  - `std::cin >> x; std::cout << x;`
- **Multiple values can be streamed at once**
  - `std::cout << "the value is " << x << '\n';`
- **Streams have a state that denotes if I/O was successful or not**
  - Only `.good()` streams actually do I/O
  - You need to `.clear()` the state in case of an error

Reading a `std::string` Value

```
#include <istream>
#include <string>

auto inputName(std::istream & in) -> std::string {
  std::string name{};
  in >> name;
  return name;
}
```

- **Reading a `std::string` can not go wrong, unless the stream is already `!good()`**
  - The content of the `std::string` is replaced
  - Maybe the `std::string` is empty after reading

Reading an `int` Value

```
auto inputAge(std::istream& in) -> int {
  int age{-1};
  if (in >> age) {
    return age;
  }
  return -1;
}
```

- **No error recovery**
- **One wrong input puts the stream into status fail**
- **Characters remain in input**

```
auto inputAge(std::istream& in) -> int {
  int age{-1};
  if (in >> age) {
    return age;
  }
  return -1;
}
```

Boolean conversion as post-read check

- **Result of `in >> age` is the istream object itself**
- **The stream object converts to bool (in `if` and loop conditions)**
  - `true` if the last reading operation has been successful
  - `false` if the last reading operation failed somehow (formatting, stream end or another problem)

```
auto readSymbols(std::istream& in) -> std::string {
  char symbol{};
  int count{-1};
  if (in >> symbol >> count) {
    return std::string(count, symbol);
  }
  return "error";
}
```

- **Result of `in >>` age is the istream object itself**
- **Multiple subsequent reads are possible**
- **If a previous read already failed, subsequent reads fail as well**

```cpp
#include <iostream>

auto main() -> int {
  size_t count{0};
  char c{};
  while (std::cin >> c) ++count;
  std::cout << count << "\n";
}
```

```
$ mycharcount < input.txt
42
$ mycharcount
12345
<CTRL-D>|<CTRL-Z>
6
$
```

- **If you write programs to read all of the input you need to terminate the input:**
  - Ctrl-D (Linux/Mac) and Ctrl-Z (Windows)
- **Press <Enter> to send the current line to the input**
  - You may edit the line before sending, e.g. delete characters

Robust Reading an `int` Value


```
auto inputAge(std::istream & in) -> int {
  std::string line{};
  while (getline(in, line)) {
    std::istringstream is{line};
    int age{-1};
    if (is >> age) {
      return age;
    }
  }
  return -1;
}
```

- **Read a line and parse it as an integer until `OK` or `EOF`**
- **Read operation in while condition acts as a "did the read work?" check**
- **Use an `std::istringstream` as intermediate stream**

```
auto readFrom(std::istream & is) -> int {
  //...
}
```

State Bit Set	Query	Entered
`<none>`	`is.good()`	initial `is.clear()`
`failbit`	`is.fail()`	formatted input failed
`eofbit`	`is.eof()`	trying to read at end of input
`badbit`	`is.bad()`	unrecoverable I/O error

- **Formatted input on stream `is` must check for `is.fail()` and `is.bad()`**
  - If failed, `is.clear()` the stream and consume invalid input characters before continue

ios_base::iostate flags			basic_ios accessors					
`eofbit`	`failbit`	`badbit`	`good()`	`fail()`	`bad()`	`eof()`	`operator bool`	`operator !`
`false`	`false`	`false`	`true`	`false`	`false`	`false`	`true`	`false`
`false`	`false`	`true`	`false`	`true`	`true`	`false`	`false`	`true`
`false`	`true`	`false`	`false`	`true`	`false`	`false`	`false`	`true`
`false`	`true`	`true`	`false`	`true`	`true`	`false`	`false`	`true`
`true`	`false`	`false`	`false`	`false`	`false`	`true`	`true`	`false`
`true`	`false`	`true`	`false`	`true`	`true`	`true`	`false`	`true`
`true`	`true`	`false`	`false`	`true`	`false`	`true`	`false`	`true`
`true`	`true`	`true`	`false`	`true`	`true`	`true`	`false`	`true`

Table from: https://en.cppreference.com/w/cpp/io/ios_base/iostate

```cpp
auto inputAge(std::istream & in) -> int {
  while (in.good()) {
    int age{-1};
    if (in >> age) {
      return age;
    }
    in.clear();  // remove fail flag
    in.ignore(); // one char
    // alt: in.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
    // ignores whole line
  }
  return -1;
}
```

good() as precondition check for successful read

```cpp
#include <iostream>
#include <iomanip>
#include <ios>
```

```cpp
auto main() -> int {
  std::cout << 42 << '\t'
            << std::oct << 42 << '\t'
            << std::hex << 42 << '\n';
  std::cout << 42 << '\t' // std::hex is sticky
            << std::dec << 42 << '\n';
  std::cout << std::setw(10) << 42
            << std::left << std::setw(5)<< 43 << "*\n";
  std::cout << std::setw(10) << "hallo"<<"*\n";

  double const pi{std::acos(0.5) * 3};
  std::cout << std::setprecision(4) << pi << '\n';
  std::cout << std::scientific << pi <<  '\n';
  std::cout << std::fixed << pi * 1e6 <<  '\n';
}
```

Complete list of includes and manipulators: https://en.cppreference.com/w/cpp/io/manip

```
#include <iostream>
#include <cctype>
auto main() -> int {
  char c{};
  while (std::cin.get(c)) {
    std::cout.put(std::tolower(c));
  }
}
```

- **A very simple program transforming its input to lower case**
  - `<cctype>` contains character conversion and character kind query functions (`std::tolower(c)`, `std::isupper(c)`)
- **`get()` and `put()` are unformatted I/O functions**
  - What happens when we use `>>` and `<<` ?
- **More in the exercises for you to experiment with!**

- **`iosfwd` contains only the declarations for std::ostream and std::istream**
  - In header files (.pph) this is usually sufficient when the streams are only used in function declarations
- **`istream` and `ostream` contain the implementation of the corresponding stream, operators**
  - Usually, these are required in source files (.cpp) when the streams are actually used in functions
- **`iostream` contains all of the above and additionally `std::cout`, `std::cin`, `std::cerr`**
  - This is only required in the source file containing the `main()` function, because only there the global standard IO variables shall be used
- **General advice: only use the minimally required header**

Summary

- **Variables can keep a value and have a type**
- **`const` makes variables single-assignment only, no changes**
- **Output can be done using `ostream`, i.e., `std::cout` and `<<`**
- **Input uses `istream`, i.e., `std::cin` and `>>` to an lvalue**
- **Streams have a state for `eof` and format errors on input**

- **No significant changes in the slides since C++17**
  - Trailing return types adjusted
  - Replaced Cevelop references
  - Note on three-way-comparison